RELATÓRIO DE PROJETO
versão 2.3

Programas desenvolvidos, testados e documentados

1.Termo de Referência

Esse relatório diz respeito ao edital **número 46**, "**OBJ-REL - Camada de Persistência Objeto-Relacional"**, publicado entre os dias 04 a 11 de maio de 2008, a ser realizado no período de maio/2008 a outubro/2008.

2.Introdução

De acordo com o descrito no termo de referência desse projeto, os frameworks utilizados pelo Interlegis permitem o armazenamento de informações tanto em banco de dados relacionais (PostgreSQL e MySQL) quanto em banco de dados orientados a objetos (ZODB).

De forma geral, quando trata-se de representar informações dentro do Plone (utilizando-se o framework Archetypes), a persistência dos dados costuma ser orientada a objetos (OO) e realizada diretamente no banco de dados orientado a objetos ZODB. Há algumas soluções do Interlegis como, por exemplo, o SAPL (Sistema de Apoio ao Processo Legislativo), que podem ser consideradas soluções "híbridas", armazenando informações em diferentes modelos (parte das informações em banco de dados relacional e parte em banco de dados orientado a objetos). No entanto, o SAPL não utiliza o framework Archetypes e, por sua vez, não está relacionado com esse projeto.

Esse relatório tratará apenas das considerações relevantes ao primeiro caso, o mapeamento de informações objeto-relacional. Nesse caso, as informações que normalmente seriam armazenadas em bancos de dados OO precisam ser mapeadas para um banco de dados relacional.

Esse segundo relatório complementa do Relatório 1 já entregue anteriormente e diz respeito a entrega dos "programas desenvolvidos, testados e documentados". Posteriormente, os relatórios 3 e 4 tratarão a respeito das demais entregas.

3. Código Fonte Desenvolvido

A solução de mapeamento objeto-relacional implementada recebeu o nome de **interlegis.sqlalchemystorage** e tem como dependência principal o módulo **archetypes.sqlalchemystorage**. O processo de instalação da solução será descrito no Relatório 3 (manual de instruções); esse relatório cobre apenas os aspectos gerais da implementação realizada.

O código fonte propriamente dito foi **completamente** desenvolvido, testado e documentado. O código fonte do projeto foi entregue por correio eletrônico enviado para o consultor do Interlegis Jean Ferri e com cópia para Petrônio Barbosa Lima de Carvalho, diretor da subsecretaria de TI do Interlegis e surpevisor nesse projeto. Além disso, o código fonte complete poderá ser baixado através do subversion, de acordo com os procedimentos a serem descritos posteriormente, no manual de instruções.

Todos os procedimentos foram testados utilizando a distribuição **Linux Ubuntu Server Edition versão 8.0.4** (sabidamente a distribuição Linux mais utilizada no Interlegis), mas os procedimentos são igualmente válidos para outras versões da

distribuição, assim como para outras distribuições baseadas no sistema de empacotamento do Debian. Além disso, o banco de dados utilizado para nos testes foi o **MySQL 5.0**, com suporte InnoDB. O banco de dados MySQL já vem sendo utilizado em outras soluções do Interlegis (como o SAPL). Além disso, a infra-estrutura descrita na especificação relatada no Relatório 1 foi cuidadosamente observada.

4. Detalhamento Técnico

O código fonte desenvolvido baseia-se fortemente na técnica de "monkey patch" (vide http://en.wikipedia.org/wiki/Monkey_patch), técnica utilizada para alterar o comportamento de um código fonte sem efetivamente ter que alterá-lo. Esse tipo de técnica é usual em linguagens dinâmicas como Python e justifica-se pela necessidade de alterar a forma como partes internas do framework Archetypes funcionam, sem a necessidade de manter uma versão própria do framework.

Todos os "monkey-patchs" desenvolvidos seguem um "design pattern" conhecido e já adotado em outros Plone Products. O código fonte da técnica segue abaixo:

```
from interlegis.sqlalchemystorage.config import logger

PATTERN = '__interlegis_sqlalchemystorage_%s__'

def call(self, __name__, *args, **kw):
    return getattr(self, PATTERN % __name__)(*args, **kw)

def call_pattern(self, __name__, pattern=PATTERN, *args, **kw):
    return getattr(self, pattern % __name__)(*args, **kw)

WRAPPER = '__interlegis_sqlalchemystorage_is_wrapper_method__'
ORIG_NAME = '__interlegis_sqlalchemystorage_original_method_name__'

def isWrapperMethod(meth):
    return getattr(meth, WRAPPER, False)

def wrap_method(klass, name, method, pattern=PATTERN):
    old_method = getattr(klass, name)
    if isWrapperMethod(old_method):
        logger.warn('Not wrapping %s.%s. Already wrapped.' %
            (klass.__name__, name))
        return
    else:
        logger.warn('Wrapping %s.%s.' %
            (klass.__name__, name))
    new_name = pattern % name
    setattr(klass, new_name, old_method)
    setattr(method, ORIG_NAME, new_name)
    setattr(method, WRAPPER, True)
    setattr(klass, name, method)

def unwrap_method(klass, name):
    old_method = getattr(klass, name)
    if not isWrapperMethod(old_method):
        raise ValueError, ('Trying to unwrap non-wrapped '
                           'method %s.%s.' % (klass.__name__, name))
    orig_name = getattr(old_method, ORIG_NAME)
    new_method = getattr(klass, orig_name)
    delattr(klass, orig_name)
    setattr(klass, name, new_method)
```

Na prática os métodos que receberam novo comportamento são "embrulhados" em um método "wrapper" que é executado no lugar do método original toda vez que esse tivesse que ser chamado.

O trecho de código que teve maior quantidade de "monkeys patches" é relacionado com a implementação da persistência de referências. Nativamente, o Archetypes

implementa a API de referências entre objetos em uma classe base, a Archetypes.Referenceable.Referenceable. Essa classe implementa métodos para consultar quais referências apontam para um objeto, adicionar e remover referências. O código abaixo mostra quais métodos receberam novo comportamento utilizando essa técnica:

```
def patch_referenceable(klass):
    klass_name = klass.__name__
    #pattern = '__interlegis.sqlalchemystorage_%s' % klass_name + '_%s__'
    config.logger.warn('Applying Referenceable patches for %s' % klass_name)
    if config.storage != 'BOTH':
        patch.wrap_method(klass, 'getRefs', ATSAReferenceable_getRefs)
        patch.wrap_method(klass, 'getReferences', ATSAReferenceable_getRefs)
        patch.wrap_method(klass, 'getBRefs', ATSAReferenceable_getBRefs)
        patch.wrap_method(klass, 'getBackReferences', ATSAReferenceable_getBRefs)
        patch.wrap_method(klass, 'hasRelationshipTo', ATSAReferenceable_hasRelationshipTo)
        patch.wrap_method(klass, 'getRelationships', ATSAReferenceable_getRelationships)
        patch.wrap_method(klass, 'getBRelationships', ATSAReferenceable_getBRelationships)
        patch.wrap_method(klass, 'getReferenceImpl', ATSAReferenceable_getReferenceImpl)
        patch.wrap_method(klass, 'getBackReferenceImpl', ATSAReferenceable_getBackReferenceImpl)
    patch.wrap_method(klass, 'addReference', ATSAReferenceable_addReference)
    patch.wrap_method(klass, 'deleteReference', ATSAReferenceable_deleteReference)
    patch.wrap_method(klass, 'deleteReferences', ATSAReferenceable_deleteReferences)
    config.logger.warn('Referenceable patches applied for: %s' % klass_name)

def patch_referencefield(field, klass):
    klass_name = klass.__name__
    #pattern = '__interlegis.sqlalchemystorage_%s' % klass_name + '_%s__'
    config.logger.warn('Applying ReferenceField patches for %s' % klass_name)
    patch.wrap_method(field, 'set', ATSAReferenceField_set(field))
    config.logger.warn('ReferenceField patches applied for: %s' % klass_name)
```

O trecho acima exibe quais métodos foram embrulhados com monkeys patchs e encontra-se no final do módulo references.py. Nesse mesmo módulo, encontramos a implementação dos patches propriamente ditos. Abaixo, segue alguns exemplos de como foi possível mudar o comportamento dos métodos para passar a persistir as informações que antes eram mantidas no ZODB e agora podem ser persistidas em um banco de dados relacional utilizando o framework SQLAlchemy.

```
def ATSAReferenceable_getRelationships(self):
    """What kinds of relationships does this object have"""
    db = get_database(self)
    rm = db.mappers['reference_catalog']
    return list(set([ref.relationship for ref in
                     db.session.query(rm).filter_by(sourceuid=self.UID())]))

def ATSAReferenceable_getBRelationships(self):
    """
    What kinds of relationships does this object have from others
    """
    db = get_database(self)
    rm = db.mappers['reference_catalog']
    return list(set([ref.relationship for ref in
                     db.session.query(rm).filter_by(targetuid=self.UID())]))
```

```
def ATSAReferenceable_deleteReference(self, target, relationship=None):
    if config.storage == 'BOTH':
        Referenceable.deleteReference(self, target, relationship)
    filter_by = dict(sourceuid=self.UID(),
                     targetuid=target.UID())
    if relationship is not None:
        filter_by['relationship'] = relationship
    for m in _get_refs(self, filter_by):
        db.session.delete(m)

def ATSAReferenceable_deleteReferences(self, relationship=None):
    if config.storage == 'BOTH':
        Referenceable.deleteReferences(self, relationship)
    filter_by = dict(sourceuid=self.UID())
    if relationship is not None:
        filter_by['relationship'] = relationship
    for m in _get_refs(self, filter_by):
        db.session.delete(m)
```

Como descrito no início do capítulo 3 desse relatório, a implementação foi dividida em 2 partes. Abaixo está detalhada a estrutura de módulos do código fonte.

```
.
|-- EXTERNALS.txt
|-- archetypes.sqlalchemystorage
|   |-- archetypes
|   |   |-- __init__.py
|   |   `-- sqlalchemystorage
|   |       |-- __init__.py
|   |       |-- config.py
|   |       |-- configure.zcml
|   |       |-- database.py
|   |       |-- docs
|   |       |   |-- CREDITS.txt
|   |       |   `-- LICENCE.GPL
|   |       |-- event.py
|   |       |-- index.py
|   |       |-- interfaces.py
|   |       |-- mapper.py
|   |       |-- storage.py
|   |       |-- templates
|   |       |   `-- add.pt
|   |       `-- tests
|   |           |-- __init__.py
|   |           |-- configure.zcml
|   |           |-- test_SAIndex.py
|   |           |-- test_indexStorage.py
|   |           `-- test_storage.py
|   |-- setup.cfg
|   `-- setup.py
`-- interlegis.sqlalchemystorage
    |-- README.txt
    |-- docs
    |   |-- HISTORY.txt
    |   |-- INSTALL.txt
    |   |-- LICENSE.GPL
    |   `-- LICENSE.txt
    |-- interlegis
    |   |-- __init__.py
    |   `-- sqlalchemystorage
    |       |-- README.txt
    |       |-- TODO.txt
    |       |-- __init__.py
```

```
    │           ├── columns.py
    │           ├── config.py
    │           ├── configure.zcml
    │           ├── database.py
    │           ├── database.txt
    │           ├── declarative.py
    │           ├── events.py
    │           ├── fss.py
    │           ├── fss.txt
    │           ├── generator.py
    │           ├── generator.txt
    │           ├── interfaces.py
    │           ├── patch.py
    │           ├── referenceable.py
    │           ├── referenceable.txt
    │           ├── sastorage_patch.py
    │           ├── sastorage_patch.txt
    │           ├── storage.conf
    │           └── tests
    │               ├── __init__.py
    │               ├── configure.zcml
    │               ├── database.py
    │               └── test_base.py
    ├── setup.cfg
    └── setup.py

11 diretórios, 54 arquivos
```